Agile Parsing in TXL

Thomas R. Dean James R. Cordy Andrew J. Malton Kevin A. Schneider

Abstract

Syntactic analysis forms a foundation of many source analysis and reverse engineering tools. However, a single standard grammar is not always appropriate for all source analysis and manipulation tasks. Small custom modifications to the grammar can make the programs used to implement these tasks simpler, clearer and more efficient. This leads to a new paradigm for programming these tools: agile parsing. In agile parsing the effective grammar used by a particular tool is a combination of two parts: the standard base grammar for the input language, and a set of explicit grammar overrides that modify the parse to support the task at hand. This paper introduces the basic techniques of agile parsing in TXL and discusses several industry proven techniques for exploiting agile parsing in software source analysis and transformation.

1. Introduction

Syntactic analysis plays a large role in the analysis and manipulation of software systems. Syntax is the framework on which the semantics of most modern languages is defined. For example, we use syntactic techniques to resolve the scoping of names and the precedence of operators in expressions. Not surprisingly, syntactic analysis forms the foundation of many source analysis and reverse engineering tools, including ASF+SDF (van den Brand,2001), DMS (Baxter,1997), Stratego (Visser,2001), REFINE (Reasoning,1992), Draco (Neighbors,1984) and TXL (Cordy,1991, Cordy,2000).

Source code analysis and manipulation tools use parsers in software comprehension, transformation and migration tasks. Analyzing and transforming source code using these tools involves crafting an appropriate source language grammar and parsing the input source code into a parse tree. Analysis and transformation rule sets then use the grammar to structure the patterns and transformations that are applied to the source code in its parse tree form.

Parsers have come a long way since the days of lex (Lesk,1975) and yacc (Johnson,1975). While LR, LL, LALR and other similar context free grammar classes and their associated parsing algorithms have strengths that lead to the design and implementation of efficient compilers, the restrictions they impose on the grammars limit their effectiveness in software analysis and manipulation tools (van den Brand,1998). Generalized parsing algorithms such as those used in ASF+SDF, DMS and TXL permit us to use grammars that are closer to the natural structure of the language (van den Brand,1998), making it significantly easier to specify analysis and transformation tasks.

Unlike compiler grammars, which are focussed on speed and syntax error detection, these parsing technologies allow for unrestricted "user level" grammars that reflect the language definition more or less directly. These parsers directly accept grammars are much closer to the "natural" structure of the language, making it easier to write new grammars, easier to write analysis patterns and transformations using "native" patterns (Sellink,1999,Sellink,1998), and easier to modify the grammar to adapt to new dialects (e.g., Borland C, Gnu

C++, Microsoft C, etc.) and embedded sublanguages (e.g., SQL, CICS, and so on).

This ability to easily adapt the grammar leads to the key idea of this paper. If we can easily modify the grammar in order to accept new dialects, sub-languages and the like, *why not exploit this capability to adapt the grammar to best suit each individual analysis tool?* This simple idea is the basis of the technique we call *agile parsing*.

The next section of the paper discusses the general concept of agile parsing. Section 3 reviews the TXL language, focussing on the features of the language that support agile parsing. Section 4 explores agile parsing in depth, presenting seven agile parsing idioms. Section 5 discusses some of our experience using these idioms and includes data on the performance of the TXL parser. Related work is discussed in Section 6 and concluding remarks are given in Section 7.

2. Agile Parsing

Agile parsing refers to the ability to use a customized version of the input language grammar for each particular analysis and transformation task. Based on a standard *base grammar* for the input language, agile parsing provides the ability to *override* nonterminal definitions on a per-task basis to modify the parse to yield an AST that makes the source analysis or transformation more efficient and convenient. The base grammar is a user-level grammar that defines the outline structure of the expected input language and its standard nonterminal categories. Base grammars are often recovered directly from language manuals, by hand or in a manner similar to Lämmel and Verhoef's semi-automation (Lämmel,2001a). The base grammar serves as the common understanding of the language that is modified on an *ad hoc* basis to suit each particular analysis task.

For each specific task, a set of *grammar overrides* are specified. Overrides modify or extend the forms of particular nonterminals of the base grammar to change the grammar to yield a parse that is tailored to the task at hand. These small changes to the grammar can make a significant difference in the simplicity, efficiency and maintainability of analysis programs. For example, by changing the grammar to conflate several similar nonterminal forms into a single nonterminal, we can often reduce a large number of different analysis rules to just one.

Because the nonterminals of the base grammar are used as representatives of language structures and concepts in the analysis and transformation rules, it is important that their names reflect the semantic role that each particular nonterminal plays in the input language. This is very similar to the generally accepted practice in procedural programming of giving constants and variables names that are representative of the real world concept they represent. In TXL base grammars, the names of the nonterminals are typically those that are found in standard reference grammars. These observations are not new (Sellink,1999,Sellink,1998), but they are relevant to the discussion of the rest of the paper.

Agile parsing is very much an evolved technique of TXL programming. Experienced TXL programmers are as likely to change the *grammar* by writing new overrides to assist in an analysis or transformation as they are to write a new rule for it. Programmers colloquially refer to this practice as *“grammar programming”*.

Figure 1 shows a traditional software analysis and transformation architecture using a single grammar. The source is parsed, one or more transforms are applied and after all transforms have been run, the output is generated. While some toolkits provide a modular architecture for organizing rule sets, saving the parse trees and allowing separate modules to read the trees and operate on them, the source is in general parse only once. All transforms must conform to the global grammar and are tightly coupled to it.

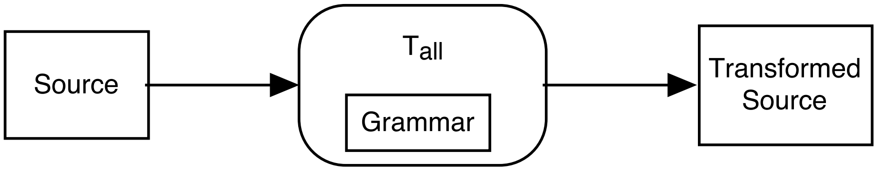

Figure 1. Single grammar transform architecture.

In this architecture the source code is parsed to a syntax tree once using a single general grammar. All stages of the tool are constrained to work using the same grammar.

Figure 2 shows an alternate structure for the same tool using agile parsing. In this approach, a different variant of the grammar can be used for each transform. The input is returned to source form after each transform. In some cases the modifications to the grammar may be minor, but in other cases the changes to the grammar (and therefore the parse tree) may be significant. The first point of this architecture is that the rule sets for each transform are not coupled by the grammar. If a rule set needs modifications to the grammar, it can have those modifications independently of the other transforms without fear of breaking them.

But there is also something more fundamental happening in this architecture. In a very real sense we are using the parser itself as a transformation engine. It transforms the tree from the results of the previous transformation to a form more suitable for the next transformation. For example, in one transform we might choose to use a modified grammar that parses the **else** clauses of **if-then-else** statements as separate statements, while the other transforms use the conventional parse of **if-then-else** statements specified in the original base grammar.

3. TXL

TXL (Cordy,1991,Cordy,2000) is a software analysis and transformation system based on the assumption that any software task can be modeled as a source-to-source transformation. TXL is a pure functional programming language specifically designed to support structural source transformation. The structure of the source to be transformed is described using an unrestricted ambiguous context free grammar from which a parser is automatically derived. While based on a top-down approach, this parser has full backtracking and ordering heuristics to resolve both ambiguity and left recursion.

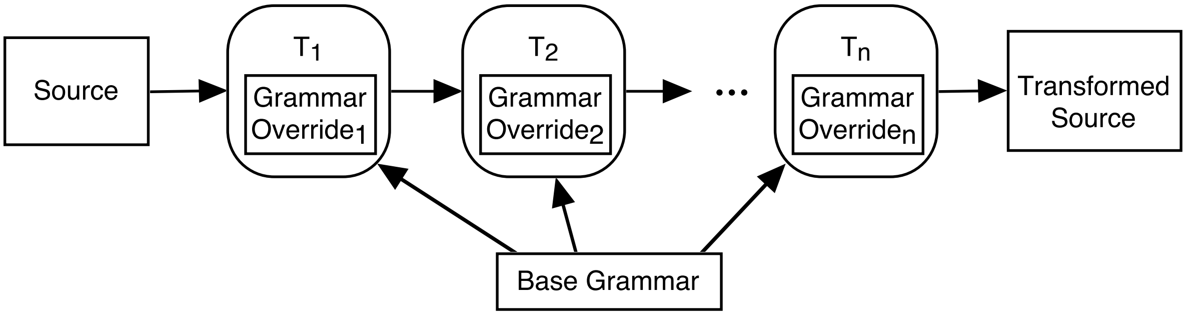

Figure 2. Multiple grammar transform architecture.

In this architecture the source code is re-parsed to a syntax tree using a custom variant of the general grammar to suit each stage of the tool.

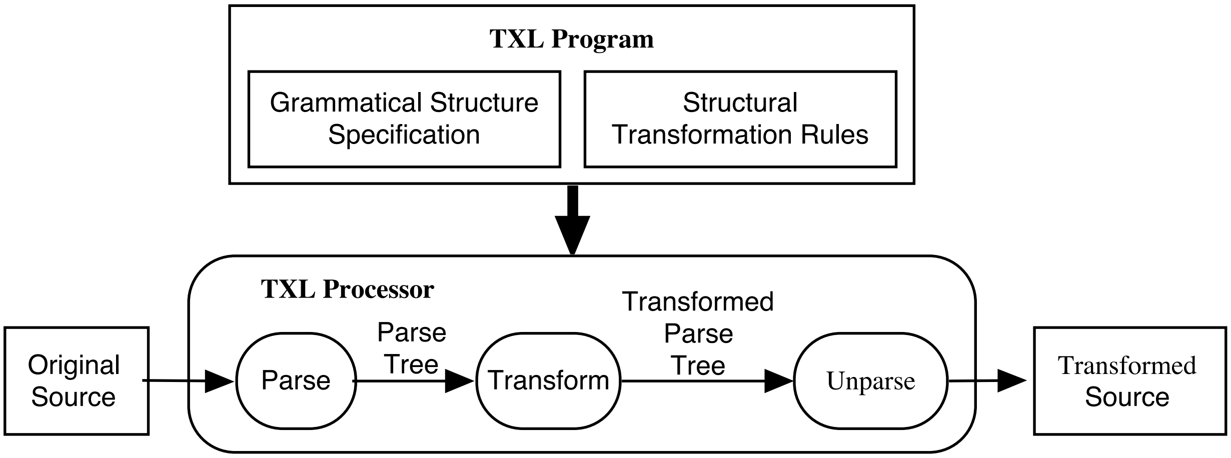

Figure 3. The TXL processor.

The transformations are described by example, using a set of context-sensitive structural transformation rules from which an application strategy is automatically inferred. Examples of our experience in using TXL in software engineering problems have been reported elsewhere (Cordy,2002).

Figure 3 shows the structure of the TXL language system. A TXL program which is comprised of a grammar and a set of rules is read by the TXL processor. A parser is derived from the grammar and the input is parsed based on that grammar. The resulting parse tree is passed to the transform engine which applies the rules transforming the input tree as they run. The last phase of the processor walks the parse tree and produces the transformed output.

One of TXL's strengths is that it uses a run-time interpretive parser. There is no separate generation step used to process the grammar and rules into a tool, and the grammar and rules can be modified and the tool rerun in seconds.

3.1 The TXL Language

A full description of the TXL language is beyond the scope of this paper. However, we will give a short overview for readers who are not familiar with the language. Figure 4 shows a subset of the TXL base grammar for a simple Pascal-like language. The special nonterminal name *program* defines the goal symbol of the grammar. Square brackets denote the use of a nonterminal in a production or rule (e.g., *[program]*). Prefixing a nonterminal symbol with the keyword ***repeat*** denotes a sequence of the nonterminal. Thus a *program* in the sample language described by the TXL grammar in Figure 4 consists of the keyword ***program***, an identifier (*[id]*) and a semicolon (;) followed by a sequence of *definitions* and a *block*, terminated by a period (.).

Vertical bars ('|') are used to separate alternatives in TXL grammar productions. For example, a *definition* in our sample language can be a *[constant_section]*, a *[variable_section],* a *[type_section]* or a *[procedure_definition].* The **list** keyword is much like **repeat**, except that the repeated nonterminals are separated by a comma. Thus the *[args]* nonterminal is defined as a list of expressions separated by commas and surrounded by parentheses.

```
define program
   program [id] ;
   [repeat definition]
   [block].
end define

define definition
      [const_section]
    | [variable_section]
    | [type_section]
    | [procedure_definition]
end define
```

```
define var_decl
    [id] : [typeName]
end define

define procedure_call
    [id] [args]
end define

define args
    ‘( [list expression] ‘)
end define
```

Figure 4. TXL Grammar for a Subset of a Pascal-like language.

*In TXL notation grammar definitions are given using the **define** statement. Each **define** gives the definition of a nonterminal and all of its alternative forms. References to other terminal and nonterminal types are enclosed in square brackets [ ] and alternative forms are separated using the traditional or bar |.*

The terminals of the grammar are *tokens*, which are referred to in the same way that nonterminals are used. In our example subset grammar, the symbol *[id]* refers to the identifier token, which defaults to a contiguous sequence of letters, digits and underscores starting with a letter or underscore (e.g. a C identifier). Language tokens can be explicitly specified using regular expressions, permitting the lexical conventions of any language to be recognized. For example, COBOL identifiers (hyphens instead of underscores) or Pascal-style strings (quote instead of backslash for escaped embedded quotes) can be defined as tokens.

3.2 TXL Support for Agile Parsing

The TXL language provides several features that provide explicit support for agile parsing. A TXL program normally begins with an **include** statement for the base grammar of the input language. The basic agile parsing feature is the idea of the nonterminal "override", which allows a given nonterminal of the base grammar to be replaced with a definition more appropriate to the task at hand. Overrides are written in TXL using the **redefine** statement [Figure 5(a)]. The semantics of an override is that the effective grammar for the tool is the original base grammar with the definition of the overridden nonterminal replaced by given redefinition, yielding a different custom parse tree intended to make the task easier.

More sophisticated overrides can use extensions of the existing nonterminal form by referring to it using the the "..." notation [Figure 5(b)]. In a TXL redefine, "..." refers to the definition of the overridden nonterminal form before it was extended by the redefine. The "..." can be used in a post-extension, in which additional alternatives for the nonterminal are added after the originals by the redefine, or as a pre-extension, in which additional grammatical forms come first and take precedence over the original forms. Extension overrides often exploit TXL's ordered ambiguity resolution. New forms introduced by the redefine can ambiguously overlap existing forms, with the semantics that forms are tried in the order specified in the effective definition, with first forms taking precedence over last forms, yielding a well defined deterministic parse.

Other features of TXL supporting agile parsing include general nonterminal polymorphism (*[any]*), nonterminal type query (*[typeof]*), programmed syntactic parse tree extraction (*[^]*) and transformation-time re-

```
include "Cpp.Grammar"

redefine function_definition
   [function_header]
   [opt exception_specification]
   [function_body]
end redefine

define function_header
   [opt decl_specifiers]
   [function_declarator]
   [opt ctor_initializer]
end define
```

(a) Replacing a nonterminal in TXL.

*The **redefine** statement gives a new definition for the redefined nonterminal which replaces the original in the base grammar to yield a different parse. In this case, the new nonterminal [function_header] is introduced to capture the entire header line of C++ functions in one piece.*

```
include "Java.Grammar"

redefine expression
     ...
   | [xmltag] [expression] [xmlendtag]
end redefine

redefine method_call
     [jdbc_call]
   | ...
end redefine
```

(b) Extending a nonterminal in TXL.

*The **"..."** notation in a **redefine** statement refers to the original syntactic forms of the redefined nonterminal in order to allow extension of the nonterminal to other forms. In cases of ambiguity, the order of alternatives determines the parse. If the "..." appears first in an extension, then the old forms are preferred. If it appears last, then the new forms take precedence, as with [jdbc_call] above.*

Figure 5. TXL Support for Agile Parsing

parsing of transformed elements (*[reparse]*). Grammar reflection (modifying the grammar on-the-fly as the result of transformation or analysis) is theoretically possible and supported by the TXL engine but not yet directly available in TXL.

3.3 An Example: Extracting RSF Facts from SQLj

Figure 6(a) shows an example snippet of Java code with embedded SQL (SQLj). The example consists of a Java function method *getComm* that returns the commission rate for a salesperson given the employee number by querying an SQL database. We would like to extract a Rigi Standard Form (RSF) (Müller,1988) relation *MethodHostVar* which identifies the host variables (i.e. Java fields) used in the embedded SQL statements in each Java method. So for the example, the desired output is:

```
MethodHostVar getComm result
MethodHostVar getComm empId
```

These relation instances document that the java fields *result* and *empId* are used in embedded SQL statements in the *getComm* method. Our TXL program will accomplish this result in two steps, first by annotating each SQLj host variable reference with its corresponding relation instance (Figure 6(b)), and then by replacing the annotated program with its embedded relation instance annotations.

Figure 7 shows a simple TXL program used to annotate the code and extract the *MethodHostVar* relation. In order to keep the example simple enough for this paper, issues such as unique identification of methods and

```
String getComm (int empId)
{
  String result;
  #sql {
    select Commission
      into :result
      from Salary
      where empNo = :empId
  }
  return result;
}
```

(a)

```
String getComm (int empId)
{
  String result;
  #sql {
    select Commission
      into
        MethodHostVar getComm result
        :result
      from Salary
      where empNo =
        MethodHostVar getComm empId
        :empId
  }
  return result;
}
```

(b)

Figure 6. Example RSF annotation of SQLj.

The input is an SQLj Java dialect program which uses SQLj statements in its methods (a). Host variable references (references to the variables of the Java program itself rather than SQL variables) are denoted in SQLj using the standard : notation (e.g., ":result" in the program above). The transformation proceeds by annotating each SQLj host variable reference with a MethodHostVar *RSF relation (b). These RSF relations are then extracted and output as the result of the transformation.*

variables have been intentionally ignored (these issues are discussed elsewhere (Dean,2001, Malton,2001)). This program demonstrates the two main agile parsing techniques commonly used in TXL programs.

The program begins by including the Java base grammar, providing the standard structural definition of the Java language. The program the includes the SQLj dialect sub-grammar. Using agile parsing grammar overrides, this grammar modifies the Java grammar to allow for the syntactic forms of SQLj, the embedded SQL extension to Java. The sub-grammar links to the parent grammar through the use of TXL redefines that extend the Java base grammar.

Following these includes the program gives the simple nonterminal definition for an *RSF_Relation,* consisting of three identifiers on a line. The *[NL]* nonterminal is a formatting instruction to TXL to insert a newline when writing the source output for the result tree.

At this point the program uses a **redefine** to extend the syntax of *[host_variable]* from the SQLj sub-grammar to allow an RSF relation to annotate the host variable. So the redefinition modifies the *[host_variable]* nonterminal to allow the conventional form of a host variable (e.g. *:result* in Figure 6) or an *[RSF_Relation]* followed by a colon (":") and an identifier (*[id]*).

The final grammar modification redefines the whole input program form to be either it was before (a Java program with embedded SQL) or simply a sequence of RSF relations. The "..." nonterminal extension notation refers to "the previous definition" of the nonterminal, that is, to its previously defined alternative forms. The intention is that the TXL transformation will transform the input Java program to an output set of RSF relations.

The main rule of the program simply invokes the rule *[annotateHostVars]* on the whole input to embed RSF relations for each host variable in the program, and then invokes the function *[replaceByRSF]* to extract all of the embedded RSF relations from the result. The rule *[annotateHostVars]* visits each Java method once. The

```
% Based on the Java base grammar
include “Java.Grammar”

% As modified by the SQLj dialect
include “SQLj.Grammar”

% Syntax of RSF relations
define RSF_Relation
    [id] [id] [id] [NL]
end define

% Override to allow RSF annotations
% on host variables
redefine host_variable
     : [id]                  % original form
  |  [RSF_Relation] : [id]   % RSF annotated form
end redefine

% Override to allow RSF only as output
redefine program
     ...                     % “as before”
                             % (i.e., a Java program)
  |  [repeat RSF_Relation]   % or a sequence of RSF
end redefine                 %   relations

% Main function - applies all rules
function main
  replace [program]
    P [program]
  by
    P [annnotateMethodHostVars]
      [replaceByRSF]
end function
```

```
% For each method declaration ...
rule annotateMethodHostVars
   replace $ [method]
      T [type] Name [id] P [parms]
      B [body]
   by
      T Name P
      B [doEachHostVar Name]
end rule

% ... annotate each host variable
%  reference with an RSF relation
%  giving the method name
rule doEachHostVar MethodName [id]
   replace [host_variable]
      : VarName [id]
   by
      'MethodHostVar MethodName VarNam
          : VarName
end rule

        % Now replace program by its e
% RSF relations
function replaceByRSF
   replace [program]
      P [program]
   construct Rels [repeat RSF_Relation
      _ [^ P] % extract all RSF_Relati
   by
      Rels
end function
```

Figure 7. Example: generating RSF relations for SQLj host variables.

This application uses agile parsing in several different ways. First, the SQLj dialect definition is integrated into the Java base grammar using grammar overrides hidden in a separate dialect grammar file (***include*** *“SQLj.Grammar”) This modified grammar is then further refined by overriding the syntax of SQLj host variable references to allow them to be annotated with RSF relations (* ***redefine*** *host_variable). Finally, the goal symbol of the grammar (program) is overridden to allow for the output of the transformation, a sequence of RSF relations (* ***redefine*** *program). The transformation proceeds by first finding every method definition in the input program (* ***rule*** *annotateMethodHostVars) and then for each such method, annotating each SQLj host variable reference in the method with a* MethodHostVar *RSF relation relating the method name to the variable name (* ***rule*** *doEachHostVar). Finally, the TXL syntactic extraction rule [^] is used to extract all sub-parse trees of nonterminal type [RSF_Relation] from the annotated program into a sequence which replaces the program itself (* ***function*** *replaceByRSF).*

pattern of the rule separates the header of the method (type, name and parameters) from the body. It then invokes the rule *[doEachHostVar]* on the body of the method, passing the method name (e.g., M) as parameter.

The rule *[doEachHostVar]* then annotates each host variable expression (e.g., : X) in the method with the RSF relation *MethodHostName*, using the method name (e.g. M) and the variable name (e.g. X) as RSF arguments (e.g., MethodHostName M X : X). The rule terminates when all host variables in the method have been annotated.

Finally, the function *[replaceByRSF]* is applied to the entire RSF annotated program. This function uses TXL's syntactic form extraction rule (*[^]*) to retrieve all of the embedded RSF relations in the annotated program (i.e., all subtrees of syntactic form *[RSF_Relation]*). The entire annotated program is then replaced by the extracted RSF relations.

This TXL program illustrates two major idioms of agile parsing in TXL. The first is the separation of embedded dialect sublanguages from the base grammar. This same technique can be used to handle embedded SQL in COBOL or the use of CICS in COBOL or C. In this way TXL is used in a way similar to the use of language modules in ASF+SDF. The second is the modification of the grammar to support the task, in this case to allow RSF relations to be used as annotations on host variables.

The rest of this paper examines by example several other agile parsing idioms we have used over years of TXL programming. This set is by no means exhaustive. Agile parsing is as deep and complex a study as any other general programming technique, and the use of it evolves with each new project.

4. Agile Parsing Idioms

Experience with programming in TXL over the years has led to several distinct grammar programming idioms exploiting agile parsing. While these techniques are well suited to TXL's execution-time interpretive parser, most of them are not limited to TXL and could easily be applied in other typed rewriting systems such as ASF+SDF. Because of limited space, the examples we examine in this paper are necessarily simplified from the form that would be used in real TXL applications. For example, references to variable names in programs may involve various qualifiers and modifiers that we have elided in our examples. We also ignore most of the details of the rules that use the techniques to solve a particular problem.

Several of the grammar changes shown in these examples could in theory be made directly when building the base grammars common to a suite of tools, if the grammar designers had sufficient foresight. However, it is important to remember that there is no way to predict the need for such changes at language grammar design time, and that making such changes to an existing language grammar itself can be a dangerous thing that may invalidate assumptions of existing tools. By exploiting agile parsing to encode grammar tuning *ad hoc* on a tool-by-tool basis while leaving the base grammar invariant, we maximize flexibility in crafting the grammar to the task while avoiding any risk of undesired side-effects.

4.1 Rule Abstraction

Even if the base grammar is generalized as suggested in Section 2, there may be distinctions that are necessary in general but unimportant for a particular application. A simple approach would be to write many separate rules to handle each of the cases. Alternatively, one can remove the distinctions by overriding the grammar, and then use a single "abstracted" rule to handle all the cases. One example of this type of problem is the identification of those variables that are used in arithmetic contexts (i.e. addition,subtraction, etc.) in the COBOL language.

The COBOL base grammar has the normal multiple levels of precedence in the expression grammar, and the nonterminal *[statement]* derives *all* of the different statements. Without grammar modification, we must write separate rules that target each level of precedence and each type of arithmetic statement (e.g. ADD statement). By modifying the grammar, we can significantly reduce the number of rules needed to extract the information. Figure 8 shows the grammar overrides to handle arithmetic statements and the single rule that is used to extract uses of identifiers in arithmetic statements.

```
% New nonterminal type that gathers all
% arithmetic statement types

define arithmetic statement
    [add_statement]
  | [subtract_statement]
  | [mult_statement]
  | [divide_statment]
  | [compute_statement]
end define

% Override existing [statement] type to
% prefer our new [arithmetic_statement]
% and allow RSF annotations

redefine statement
      [repeat RSF_Relation]
      [arithmetic_statement]
   |  ...
end rule
```

```
% Now one rule can be used to annotate
% all arithmetic statements, instead o
% having a rule for each different kin

rule annotateArithStatements Prog[id]
 replace [statement]
    Arith [arithmetic_statement]
 construct ArithIds [repeat id]
    _ [^ Arith]
 construct Rels [repeat RSF_Relation]
   _ [buildRSF 'Arith Prog each ArithIds
 by
    Rels
    Arith
end redefine
```

Figure 8. Example of rule abstraction: annotating arithmetic statements.

*Agile parsing is used to override the [statement] nonterminal to prefer the new nonterminal type [arithmetic_statement] over the existing alternatives. This override exploits TXL's ordered ambiguity resolution since the existing alternatives for [statement] (referred to by the "..." notation in TXL overrides already include the individual arithmetic statement types. Because the new type appears as the first alternative, it will be the chosen parse for all arithmetic statements. [**repeat** RSF_Relation] allows for any number of RSF relations to be prepended to [arithmetic_statement]'s. The new parse gathers all arithmetic statements into the one nonterminal type [arithmetic_statement], allowing us to use one abstracted rule to to the annotation rather than a separate rule for each kind of arithmetic statement.*

The grammar modifications, shown on the left hand side of the figure, accomplish two things. The first is that the five COBOL arithmetic statements are grouped under a single nonterminal called *[arithmetic_statement]*. The second is that the *[statement]* nonterminal is changed to recognize arithmetic statements before other statements. This grammar is ambiguous since the five statements characterized as arithmetic statements can be also reached through the other (old) alternatives of the nonterminal *[statement]*. The TXL parser resolves such ambiguities by always choosing the first matching alternative, yielding an efficient and deterministic parse.

The right hand side of Figure 8 shows how the modified grammar is used by the rule *[annotateArithStatements]*. The rule is called with the COBOL program name as a parameter (*Prog*) and visits each arithmetic statement extracting the identifiers from the statement. The function *[buildRSF]* is used to build an RSF relation *Arith* for each of the identifiers. The changes to the grammar allow us to write one rule to accomplish this, where the base grammar would require five separate rules. In an actual application, the RSF relations generated by *[annotateArithStatements]* might be gathered together and output by a rule or function similar to *[replaceByRSF]* from Figure 7, or might be used in place directly in subsequent analysis rules.

4.2 Grammar Specialization

Grammar productions, like procedures, are often reused when the same concept is reused in a grammar. For example, the same nonterminal may be used for all references to names of variables. While this may be useful

```
define var_decl
   [list decl_name] : [type]
end define

define factor
    'not [factor]
  | [ref_name] [opt arguments]
end define
```

```
define decl_name
  [name]
end define

define ref_name
  [name]
end define
```

Figure 9. Grammar specialization in the base grammar.

This fragment of the base grammar for a Pascal-like language distinguishes declaring and referencing instances of variable names syntactically using different nonterminal types for each. Both kinds of instances have identical syntax in this language since they both derive [name].

for most analysis and manipulation tasks, for some tasks, different distinctions may be more appropriate.

Figure 9 shows an example of how a grammar may distinguish between the declaration and use of variables in a Pascal-like language. We show only the subset of the grammar involving variable declarations and variable and function references. In the places in the grammar where a name is declared, the *[decl_name]* nonterminal is used, while where a name is referenced, the *[ref_name]* nonterminal is used. Both of these symbols derive the same nonterminal, *[name]*, which in turn derives whatever a name is in the language. This permits rules to separately target declarations and references to names. This style is often used when building the base grammar for a language.

An example *ad hoc* use of this technique is to use agile parsing overrides to further refine the grammar to syntactically distinguish between references that may modify variables and references that cannot modify variables in a way corresponding to the distinctions made in the software schemas of (Lamb,1992) and (Lethbridge,2001). The left side of Figure 10 shows some of the grammar overrides for such a program. We define the nonterminals *[get_ref]* and *[put_ref]*, both of which derive the nonterminal *[ref_name]*, which is defined as it was in Figure 9. Uses of *[ref_name]* in the grammar are redefined to use the appropriate nonterminal. For example, the assignment statement, which modifies the left hand side of the assignment operator is redefined to use the *[put_ref]* nonterminal and factor, which represents a read access to a variable as a get reference (*[get_ref]*). The grammar treats a function call as a *[get_ref]* of the function. Of course any other uses of *[ref_name]* in the grammar must also be similarly specialized.

The right side of Figure 10 shows a TXL program that uses these overrides to generate the RSF instances for the *PutRef* relation for functions. The initial pattern match in the rule matches any function that has not yet been annotated. The key is the TXL syntactic extraction operator (*[^]*). In this rule, it extracts all of the instances of *[put_ref]* from the body of the function. The function *[buildRSF]* used in this example is similar to the one used in Figure 8, but takes *[ref_name]*s as parameters instead of *[id]*s. An obvious extension of this program would be to use agile parsing to abstract the cases for both functions and procedures into one rule using the rule abstraction technique of section 4.1.

4.3 Grammar Categorization

Grammar categorization refers to modification of the grammar to make finer distinctions appropriate to the task at hand. One example of the categorization idiom is the **typedef** problem in C that we discussed section

```
% New refined kinds of [ref_name]'s
% for modifying and on-modifying references
define get_ref
   [ref_name]
end define

define put_ref
   [ref_name]
end define

% References on the left of assignments
% are modifying
redefine assignment_statement
   [put_ref] := [expression]
end redefine

% References within expressions are not
redefine factor
    'not [factor]
   | [get_ref] [opt arguments]
end redefine

% Allow RSF annotations on functions
redefine function_definition
   [repeat RSF_Relation]
   'function [decl_name] [opt parms] : [typ
      [repeat definition]
      [body]
end redefine
```

```
% Rule to annotate every function with an
% RSF relation giving its modification set
rule annotateFunctions
   replace [function_definition]
     'function Fname [decl_name]
          Parms [opt parms]
          : RetT [type]
       Defs [repeat definition]
       Body [body]
    % Extract all of the [put_ref]'s from
    %   the function body as a sequence
    % "_" is the TXL notation for an
    %    initially empty sequence
    construct RefNames [repeat put_ref]
      _ [^ Body]

    % Use a subrule to turn the sequence
    %   into a set of RSF relations
    % "each" invokes the subrule once for
    %   each element in RefNames
    construct Rels [repeat RSF_Relation]
      _ [buildRSF 'PutRef Fname each RefNam
    by
      Rels
     'function Fname Parms : RetT
       Defs
       Body
end rule
```

Figure 10. Grammar specialization using agile parsing.

In this example the grammar has been further specialized to syntactically distinguish modifying references ([put_ref]'s) from readonly references ([get_ref]'s) using agile parsing overrides. This allows the simple rule [annotateFunctions] to annotate each function definition with PutRef *RSF relations giving the set of variables modified by the function.*

1.2. Although the standard C grammar does not make the distinction, it is possible to craft the grammar nonterminals and productions to categorize variable declarations and type definitions using **typedef** into separate nonterminal forms. In this particular case the grammar would still be unambiguous, because one branch of the grammar requires the keyword **typedef**, while the other branch would not permit **typedef** to occur.

A more interesting use of grammar categorization is shown in Figure 11. In this example, an ambiguity is deliberately introduced into the nonterminal *[method_call]*. As mentioned in section 5.2, the TXL parser resolves ambiguities by choosing the first listed choice that matches. The redefinition of *[method_call]* inserts a new alternative that is attempted before any of the original alternatives of the base grammar Thus calls to methods with JDBC names will be parsed as *[jdbc_call]*, while other method calls will continue to be parsed as before. Figure 11 also shows a sample rule that might use the grammar. This rule visits only method calls that have been categorized as JDBC calls, and then invokes any further analysis or transformation rules it likes on only those calls. This is a somewhat simplistic example, in that any method with the same name will be classified as a JDBC method cal, but it suffices to illustrate the technique.

```
redefine method_call
    [jdbc_call]
  | ...
end redefine

define jdbc_call
   [jdbc_name] [arguments]
end define
```

```
define jdbc_name
    'createStatement | 'prepareStatement
  | 'executeUpdate | 'executeQuery | 'getF
end define

rule processJdbc
  replace $ [method_call]
     JDBC [jdbc_call]
  by
     JDBC [doWhatever]
end rule
```

Figure 11. Grammar categorization using agile parsing.

Overriding [method_call] with the new alternative [jdbc_call] first will cause the parser to prefer the [jdbc_call] to parse calls to the specified [jdbc_name]'s. We can then take advantage of this parse to efficiently target analysis and transformation rules at [jdbc_call]'s only.

In both grammar specialization and grammar categorization we use the parser to assign different types to elements that can be identified syntactically. The difference is that in specialization we are distinguishing the *uses* of a nonterminal that is used in more than one context in the grammar. For example, *[ref_name]* may be used in both expressions and for the left hand side of assignment statements. For our purposes this may be overly abstract, and we override the grammar to make grammatical distinctions between the uses of the nonterminal in the different contexts.

In the case of grammar categorization, we have a nonterminal *definition* which is itself over general for our purposes. For example, the *[declaration]* nonterminal of C makes no distinction between type and variable declarations. If we are doing type analysis, this distinction may be very important to our task. In such cases we override the nonterminal itself to grammatically distinguish between the different forms of interest.

4.4 Union Grammars for Translation

When automatically translating between two languages, the grammar for the conversion program must be able to work with both languages. In TXL, this normally means that the grammars must be combined, since TXL transformation rules are constrained to be homomorphic (type preserving) in order to guarantee a well-formed result. If the input and output grammars are similar, they can be combined at each level where they match. For example, when translating Pascal to C, both languages are block/statement/expression based languages. We could combine the grammars at the global declaration level, the procedure level, the statement level and the expression level. As mentioned earlier, we always assume the input is syntactically valid - since the mixed grammar will allow mixed programs as input, this assumption is important here.

Figure 12 shows a greatly simplified example of how grammars can be combined at several levels using agile parsing. We redefine the nonterminal *[program]* to be a Pascal or a C program. The two grammars are rejoined at the nonterminal *[decl]* since a C program is a sequence of declarations and a Pascal program is a sequence of declarations followed by a block (the main program).

The nonterminals *[begin_or_brace]* and *[end_or_brace]* handle the minor differences between Pascal and C blocks as does the minor difference of the presence or absence of the then keyword in if statements. The new definition of the nonterminal *[block]* is a merge of the Pascal and C definitions which allows local declarations.

```
% Start with both base grammars
include "Pascal.Grammar"
include "C.Grammar"

% In the union we accept either
% kind of program
redefine program
    [pascal_program]
  | [c_program]
end redefine

define pascal_program
  'program [id] [file_header]
    [repeat decl]
    [block] '.
end define

define c_program
  [repeat decl]
end define
```

```
% Either kind of block
redefine block
  [begin_or_brace]
    [repeat decl]
    [repeat statement]
  [end_or_brace]
end redefine

define end_or_brace
  'end | '}
end define

define begin_or_brace
  'begin | '{
end define

% Either kind of if statement
redefine if_statement
  'if [expression] [opt 'then]
    [statement]
  'else
    [statement]
end redefine
```

Figure 12. Pascal / C union grammar.

In a union grammar, we exploit agile parsing to join the two grammars at appropriate points in the linguistic structure to form a grammar that allows either language's syntax. These union structures are then used as targets for transformation rules going from one language's syntax to the other.

Figure 13 shows two alternative rules to translate blocks between the languages. The first translates the block as a unit, while the second translates the difference between the beginning and end markers for the blocks separately. While this example is very simplistic, it serves to demonstrate both styles. In practice, the first (structural) style would be likely be used when translating entire statement elements such as loops, while the second (lexical) could be used to handle minor differences such as the optional "then" in the if statement.

```
rule pascalToCBlock
  replace [block]
    'begin
      Decls [repeat decl]
      Stmts [repeat statement]
   'end
  by
    {
      Decls
      Stmts
    }
end rule
```

```
rule beginToBrace
  replace [begin_or_brace]
     'begin
  by
      {
end rule

rule endToBrace
  replace [end_or_brace]
    'end
  by
    }
end rule
```

Figure 13. Two alternate Pascal to C block translation strategies.

Both strategies take advantage of the union grammar shown in Figure 12. In the solution on the left, the translation is done using one rule to transform each block structure. In the solution on the right, a pair of rules does the translation in more lexical fashion without regard to structural context.

When languages are farther apart, an alternative translation technique that exploits agile parsing more extensively is used. In this technique, the translation is broken up into multiple independent translation programs (rulesets) each of which performs a small part of the overall translation. The first program includes the base grammar for the input language (only) and uses grammar overrides only for the few changes it will make. The second translation uses the grammatical overrides from the first and adds overrides of its own for the further changes it makes. This obviously works well when the grammar overrides are independent of one another. For example, one pass might translate loop constructs while another may handle I/O statements (COBOL and PL/I have built in statements specifically for I/O). It also works well when dependent overrides are used, although the dependent overrides sometimes impose an order on the application of the rule sets. For example, translating conditional loops and translating relational expressions impose an ordering since translating a particular variant of the loop may involve negating the loop condition

At some point in time, it becomes difficult to define independent overrides and the cognitive overhead of managing the dependent overrides becomes significant. At this point in time, a new intermediate grammar is derived by excluding the translated features of the input language and including the new features of the target language that have been used. This provides a new base grammar for subsequent transforms to override. Several intermediate grammars may have to be written depending on the distance between the source and target language.

This technique has enormous advantages when developing complex language translations. Because each of the translation passes has only one responsibility, independent (except for ordering) of the other passes, translations for language features can be developed independently and in parallel. Moreover, each pass has a precise and simple independent specification, allowing for thorough independent testing and verification and reducing the chances of error.

One initial prototype for translating COBOL to Java had seventy-two passes of this kind. After two months of planning by one of the authors of this paper, the prototype was built in less than 2 months by 6 developers, not all of whom were assigned to the prototype full time. The prototype translated approximately sixty thousand of lines of code as a proof of concept for a bid to convert several million lines of code. The parallel development provided by this approach paid large dividends in time and effort. For example, three different transforms for variants of perform statements and a transform for if statements were written in parallel by different developers in approximately three days.

Writing the intermediate results to source text between transformations is an important feature of this approach. Source text isolates the changes to the grammar minimizing the dependency between rulesets while preserving TXL's strong typing within each transform. A given language feature can be parsed several different ways during the translation process, allowing different passes to employ different agile parsing techniques. It also allows the grammar used to generate a given target language feature to differ from the grammar used to parse that feature in a subsequent pass. For example, COBOL relational expressions provide abbreviated expressions and condition variables, neither of which are supported in Java. The grammar used to merge the concepts was substantially different from the conventional Java expression grammar.

Returning to text between passes also allows transforms to be developed and tested independently even when the output of one pass is required as input to the next. Since the input and output of each pass is source text, the developer can easily create by hand test cases to specify and validate a transform, even before previous transforms that create its input have been implemented. The TXL parser is fast enough that the overhead of repeated parsing is not a significant factor in translation speed (See statistics in Section 5).

The seventy-two pass COBOL to Java prototype is probably representative of the upper range of this technique. However, we would like to point out that there is a difference between the development of a special purpose program for a one time contract and a software system designed for a long term service. In our case, this was a special purpose program to transform a single COBOL system in a CICS environment to Java in an application server environment. The dominant business concern was development time, not performance or maintainability. Once other customers were found for the service, the system would be substantially simplified by merging transforms, reducing the number of intermediate grammars and overrides, and tuning performance.

4.5 Markup

For several transformation projects we found it convenient to separate the identification of the code features to be transformed from the actual transformation. For example, in LS/2000 (Dean,2001), the identification of Year 2000 sensitive code was separated from the actual remediation of the code. Since it was important that no Year 2000 bugs escape identification, and a reasonable number of false positives could be tolerated, the identification phase was very aggressive. That is, all situations (e.g. inequality expressions, arithmetic and sort keys) where date containing year digits might pose a Year 2000 risk were identified. As a result, some Year 2000 safe code was identified as potentially unsafe.

The transformation phase was correspondingly conservative, transforming only identified cases that it was certain of. The LS/2000 client could then manually fix any Year 2000 problems that were not transformed as long as they were identified. This mixed strategy meant that any false positives identified were in general not automatically remediated, saving the client from having to undo changes made by the tool. The identification and transformation programs communicated through the use of source markup of Year 2000 “hot spots” (Cordy, 2001).

Figure 14 shows an example XML markup of source code identifying a condition that leads to an abnormal termination. Such a markup is useful when tracking down unexpected errors.

Source markup can be accomplished using TXL agile parsing in two different ways: grammar based markup and polymorphic markup. Figure 15 shows an example of grammar based markup. In this method, we override the base grammar to allow markup on those elements of the grammar that we are interested in. The right hand side of Figure 15 shows a sample rule that might introduce the markup. The **skipping** statement in TXL prohibits the rule from searching inside of subtrees rooted at the given nonterminal. In this case, the skipping statement prevents the rule from going into an infinite loop. The rule replaces an expression that meets some condition with a marked up expression containing the same expression. By prohibiting the rule from matching inside of expressions, the new subexpression is not matched again.

The information used to identify which features to markup (e.g. which expression) can come from several places. It can come from analysis of other features in the program, or it can be communicated in fact from from global analysis done over several programs (Dean,2001). This technique proved to be so useful, that it was implemented in a higher level language, HSML (Cordy,2001).

```
f = fopen(filename, ”r”);
if ( <ERROR_CONDITION> f == NULL </ERROR_CONDITION> ) {
    fprintf(stderr, ”could not open %s for input\n”, filename
    exit(ENOFILE)
}
```

Figure 14. Example XML markup of C code.

```
% The C base grammar
include "C.Grammar"

% Simple grammar of markup tags
define startmark
   < [id] >
end define

define endmark
   </[id] >
end define

% Extend expressions to allow for markup
redefine expression
       ...
    | [startmark] [expression] [endmark]
end redefine
```

```
% Mark up all expressions that meet s
% semantic condition
rule annotateExpression
    % Avoid infinite markup
    skipping [expression]
    % Markup each expression ...
    replace $ [expression]
        E [expression]
    % ... that meets some condition
    where
        E [meetsSomeCondition]
    % ... as interesting.
    by
        <interesting> E </interesting
end rule
```

Figure 15. Grammar-based expression markup in C.

In this strategy each nonterminal that we want to be able to mark up separately extended using agile parsing overrides to allow for markup, as shown for [expression] above. Using this strategy, markup rules must be careful not to re-mark an already marked-up [expression]. The TXL ***skipping*** *clause helps with this by preventing rules from searching inside the specified nonterminal. The $ in the* ***replace*** *clause is a TXL optimization hint asking the tree search engine to use a one-pass strategy. The* ***where*** *clause invokes a subrule that uses other pattern matches and semantic conditions to test for the properties we're interested in.*

Markup was one of the main reasons that the polymorphic nonterminal *[any]* was introduced into TXL. Figure 16 shows a modified version of Figure 15, based on the TXL manual (Cordy,2000). Instead of overriding *[expression]* to permit markup, the *[any]* type is used to indicate that any nonterminal is permitted to be marked up, not just *[expression]*.

In this example there are two changes to the rule *[annotateExpression]*. The first is that it calls the function *[doMarkup]* to add the markup to the input. The second is that the **skipping** clause now indicates that the nonterminal markup is not to be traversed when looking for a match.

The function *[doMarkup]* uses nonterminal polymorphism (*[any]*) in its pattern to match any nonterminal node. In this case it matches the [expression] selected by *[annotateExpression]* and allows us to replace the expression nonterminal with a marked up version of itself. While they can be useful in agile parsing, polymorphic rules are generally discouraged in TXL programming since they intentionally violate the type constraints imposed by the grammar.

4.6 Semiparsing

This agile parsing technique is essentially the same as the "island" and "lake" grammar technique of van Deursen and Kuipers (van Deursen,1999) and Moonen (Moonen,2001, Moonen,2002). It can be used both to ignore sequences of input tokens in some contexts, and to pick out items of interest in a stream of otherwise uninteresting input tokens. Both versions of the technique were used in LS/2000 (Dean,2001) to deal with embedded SQL.

```
% C base grammar
include "C.Grammar"

% Polymorphic markup grammar
define startmark
    < [id] >
end define

define endmark
    </[id] >
end define

define markup
   [startmark]
     [any]
   [endmark]
end define

% As before, find all interesting
% expressions and mark them up

rule annotateExpression
    skipping [markup]
    replace $ [expression]
      E [expression]
    where
      E [meetsSomeCondition]
    by
      E [doMarkup 'interesting]
end rule
```

```
% But this time do it using a generic
% function that we can use to mark up
% anything at all

function doMarkup Tag [id]
  % We can mark up anything at all
  replace [any]
    Any [any]

  % Make the marked-up version
  construct Markup [markup]
    < Tag > Any </ Tag >

  % Type convert it back to generic ..
  deconstruct Markup
    MarkupAny [any]

  % ... and replace
  by
    MarkupAny
end function
```

Figure 16. Polymorphic grammar and rules for markup.

In this case the type [markup] exploits TXL's grammar polymorphism to allow for markup on any nonterminal at all ([any]). The function [doMarkup] exploits this polymorphism to mark up anything it is applied to with the given tag. This technique is particularly useful in applications when many different nonterminals are being marked up, because it does not require overrides.

While any generalized parsing algorithm supports Moonen's technique, TXL's parser has an extra feature to make a variant of the technique easier to use. Figure 17 shows as subset of a TXL grammar that recognizes embedded SQL. The key feature in this grammar is the nonterminal modifier **not**. The TXL expression *[**not** end_exec]* tells the parser that the following grammatical form cannot match the same sequence of tokens that the nonterminal *[end_exec]* matches. [**not**] is essentially a lookahead check; it does not consume any tokens from the input. This acts as a guard preventing the parser from consuming non-SQL tokens in error. To extend Moonen's metaphor, it can be thought of as a breakwater that prevents the lake from consuming shoreline. The predefined nonterminal *[token]* matches any token but a keyword, while the nonterminal *[key]* matches any keyword token.

The definition of the *[host_variable]* nonterminal extracts host variables out of the otherwise unparsed SQL statements. This would permit analysis rules to check if variables are referenced in SQL statements while not having to deal with the details of the SQL syntax. The *[token_or_key]* nonterminal is a TXL idiom used when a grammar author wants to consume one or more arbitrary lexical tokens.

```
% Begin with Cobol
include "Cobol.Grammar"

% Extend to allow for SQL statemen
redefine statement
    ...
  | [sql_statement]
end redefine

define sql_statement
    EXEC SQL
      [repeat sql_item]
    [end_exec]
end define

define end_exec
    END-EXEC
end define
```

```
% Use lake and island parsing to parse o
% the parts of the SQL we're interested
% (i.e., host variable references) and
% ignore the rest
define sql_item
    [host_variable]
  | [water]
end define

define host_variable
    : [ref_name]
end define

define water
    % Bounded by the END-EXEC shoreline
    [not end_exec] [token_or_key]
end define

define token_or_key
    % TXL idiom for "any lexeme"
    [token] | [key]
end define
```

Figure 17. Semi-parsing ("lake and islands") grammar for embedded SQL.

We begin by extending the [statement] nonterminal to allow for the new form [sql_statement]. Since we are only interested in [host_variable] references embedded in the SQL, we avoid parsing the complex SQL syntax using semi-parsing to recognize the embedded [host_variable] "islands" and slough off the uninteresting "water" (arbitrary other input) until we hit the end of the SQL statement (END-EXEC).

4.7 Data Structure Grammars

There is no reason why every nonterminal in the grammar should be reachable from the goal symbol. A common technique in TXL programs is to provide separate grammars to hold auxiliary data structures not part of the main input. Remember that the grammar defines the type space of a TXL program.

TXL can read from more than one file, and the input from the second file need not be parsed with the same grammar. Figure 18 shows a generalized markup program in TXL. The main input is a Java program in which each name (declaration or reference) has been annotated with a string containing a unique identifier. The form and use of this annotation is discussed elsewhere (Dean,2001, Guo,2003). For the purpose of the TXL program in Figure 18, each declaration and reference of a name has string embedded within it which contains the unique identifier. A second file contains a list of unique identifiers and the tags that are to be applied to them. This file is generated from some other analysis program. The file is read by the main rule and is parsed as a sequence of *[markItem]*s, each of which is a *[stringlit]* paired with an *[id]*.

The work is done by the rule *[doMarkup],* which visits each reference (*[ref_name]*). The unique identifier (the string literal) is extracted from the reference using a pattern match (the first **deconstruct** statement). the second **deconstruct** statement is a pattern match that searches the sequence of *[markitems]* for one that contains the unique identifier. If a matching pair is found, the matching [id] is bound to the variable *MarkupTag* for use in the rest of the rule. The convention of using a searching **deconstruct** on a sequence of pairs such as

```
% The Java base grammar and unique
% naming overrides
include “Java.Grammar”
include “JavaUID.Grammar”

% Simple grammar of markup tags
define startmark
   < [id] >
end define

define endmark
   </[id] >
end define

% Extend refs to allow for markup
redefine ref_name
      ...
   | [startmark]
       [ref_name]
     [endmark]
end redefine

% Syntax of items in second input
define markItem
   [stringlit] [id]
end define
```

```
% Main function - main input is parsed as
% Java [program], second input as a sequer
% of [markItem]’s

function main
    % Main input - a Java program
    replace [program]
      P [program]

    % Second input - a file of markup
    % directives of the form
    %    “YY DATEREC Foo.cob” YYDate
    construct MarkTable [repeat markItem]
       _ [read “MarkList”]

    % Do all the markups specified by the
    % directives
    by
      P [doMarkup MarkTable]
end function

% Mark up references as directed

rule doMarkup MarkTable [repeat markItem]
    % Avoid infinite markup
    skipping [expression]

    % For each ref_name ...
    replace $ [ref_name]
        Ref [ref_name]

    % ... get its unique name string ...
    deconstruct * [stringlit] Ref
        RefUniqueName [stringlit]

    % ... and if there is a markup directi
    % for it in the table ...
    deconstruct * [markItem] MarkTable
        RefUniqueName MarkupTag [id]

    % ... then mark it up with the given t
    by
        <MarkupTag> Ref </MarkupTag>
end rule
```

Figure 18. General markup program using a second input with a different grammar.

*This is an example of using two separate unrelated grammars, one to handle the main input (in this case a uniquely named Java program) and one to parse a secondary input (in this case a table of markup directives). The program parses each using its own grammar goal ([program] for the uniquely named Java, and [**repeat** markItem] for the table of directives). The [doMarkup] rule searches for each variable reference in the program ([ref_name]), uses a deep pattern match (**deconstruct** * [stringlit]) to find its unique name string, and then uses another deep pattern match (**deconstruct** * [markItem]) to see if the reference’s unique name is in the table of directives and if so what its markup tag should be. Deep pattern matches (**deconstruct** *) search the given parse tree for an instance of a subtree matching the given pattern.*

[markItem] is a recurring technique in TXL programs that implements an associative lookup table. Lookup tables can be read from files, constructed by rules, or be coded as constants in a TXL program.

Other data structures can be built. They can be entirely isolated from the main grammar, or they can share nonterminals with the main grammar. Thus the grammar section of a TXL program can be considered to define a forest of trees with possibly shared branches and leaves rather than simply one tree.

5. Experience

The core concept in all of these agile parsing techniques is that of a base grammar coupled with a set of task specific overrides. The base grammar is crafted to be appropriate for general transformations and to be easily modified for custom transformations. Each program then uses a set of nonterminal overrides to enlist the power of the parser to structure the input in the most convenient way for the task at hand.

The agile parsing techniques discussed in this paper have been extensively proven in large scale industrial use. All of the techniques were applied in the LS/2000 (Dean,2001) and LS/AMT tools developed at Legasys Corporation. Together these two tools have analyzed and transformed more than 4.5 billion lines of COBOL, PL/I and RPG source code. While individual projects are confidential, some of the subjects of these projects were :

- Automated Year 2000 analysis and remediation (COBOL, PL/I, RPG)
- Automated Language Translation (FORTRAN to Java, COBOL to Java)
- Automated migration from a character terminal environment to an enterprise messaging environment
- Automated migration from a character terminal environment to a three tier web based environment
- Performance analysis of a mutli-step mainframe batch program
- Analysis of decision points leading to abnormal termination.

Since these tools were all being used in an industrial setting, performance of the tools was a serious concern. Our profiling of the time spent in these processes indicates that in spite of the fact that agile parsing techniques were used extensively, parsing was never a significant fraction of the execution time. As a matter of fact, significant performance gains in the runtime of our tools were experienced almost every time the rule sets were simplified using one of these techniques. For example, the unique renaming transformation of the LS/2000 system was sped up by factor of more than 10 by introducing a small number of customized grammar overrides to simplify and tune the renaming ruleset, with no measurable increase in parse time.

The techniques described in this paper depend heavily on the generality and flexibility of the parser. TXL's parser gains its agility using an extremely naive interpretive full-backtracking top-down parser. Although the theoretical worst-case performance of this parser on a worst-case context-free grammar is exponential, in practice real programming languages (and hence their grammars) are dominated by linear productions such as statement and declaration sequences (in TXL, *[**repeat** statement], [**repeat** external_declaration],* and so on), with the more general productions limited to local structures such as expressions. These practical realities have the effect of reducing TXL parse performance to effectively linear, because observed practical limits on the length of coded expressions (i.e., at most a couple of lines) bound the cost of local backtracking to a constant (say, *k*), which when multiplied by the LL behaviour of the sequential higher level productions *N* yields an observed complexity of *k*N*, which is the measured behaviour of the TXL parsers for C++, Java, COBOL, RPG, PL/I and most other languages.

This approximately linear behaviour can be validated experimentally. On an 800 MHz PowerPC, using sources from the open source *jedit* application, the FreeTXL (TXL,2003) parser takes 0.24 seconds to parse and pretty-print a 1,000 line Java source file, 1.35 seconds for a 10,000 line source file and 11.74 seconds for a 100,000 line source file. The corresponding numbers for parsing and pretty-printing C++ are 0.17 seconds for a 1,000 line C++ source file, 1.80 seconds for 10,000 lines and 18.18 seconds for 100,000 lines (about 5,000 lines per second).

Of more relevance to this paper is performance when exploiting agile parsing. In (Cordy, 2003) we have demonstrated a generalized generic selective AST markup technique using the polymorphic markup method of Figure 16. On an 800 MHz PowerPC, this technique takes only 0.52 seconds of CPU time to selectively mark up all expressions and function declarators in the 800 line standard open source *groff.cpp* program, compared to 0.16 seconds of CPU time to parse and pretty print it only. More importantly, selective XML AST markup exhibits the typical approximately linear performance of the TXL parser when the size of the input is scaled up. On the same machine, marking up all of the program structure AST nodes (declaration and statement categories, etc.) for a 2,200 line Java source file taken from the open source *jedit* application takes 5.33 seconds, and marking up the entire 22,036 line source of *jedit* as a single source file takes 41.5 seconds.

Figure 19 shows the architecture of the LS/2000 automated Year 2000 analysis and remediation system for the languages COBOL, PL/I and RPG. Agile parsing techniques are used extensively in the *Import*, *Design Recovery Hot Spots* and *Version Integration* subsystems. All of the subsystems use a common base grammar, although a different base grammar is used for each language. All of the base grammar require that the declarations and references to named entities are annotated with a unique identifier ("UID") and that all record defini-

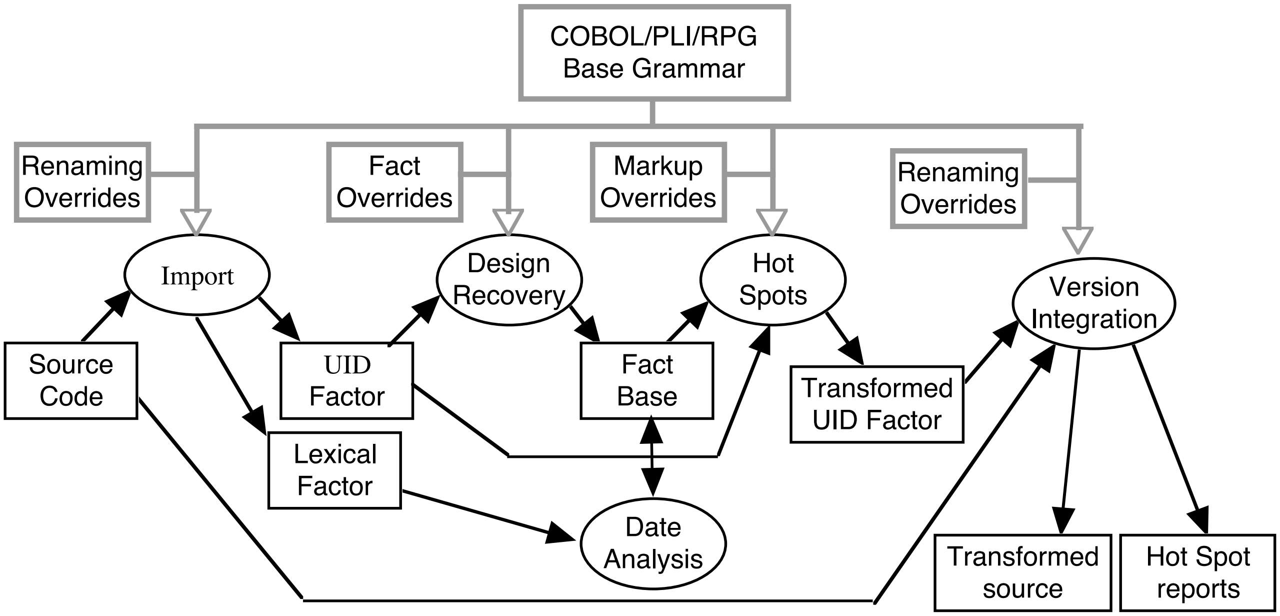

Figure 19. LS/2000 system architecture

In the LS/2000 system, the subsystems Import, Design Recovery, Hot Spots and Version Integration share a common base grammar. Each of the subsystems uses agile parsing techniques to modify the base gramma to make the task simpler. The Import and Version Integration phases use Renaming Overrides to assist in transforming the source into an internal form and back. Design Recovery uses Fact Overrides to assist in ex tracting model information from the source code, and Hot Spots uses Markup Overrides to help in identifying and transforming Year 2000 sensitive code.

tions be fully disambiguated using brackets (COBOL and PL/I use a context sensitive method of determining record structure and nesting). Almost all of the agile techniques discussed in this paper were used in LS/2000.

The *Import* and *Version Integration* subsystems use a set of grammar overrides (*Renaming Overrides*) that make UID annotation and brackets optional so that raw source code can be transformed to the internal form and back. *Import* is responsible for adding unique names and brackets while *Version Integration* removes them. The *Design Recovery* and *Hot Spots* components also have their own different variants of the main grammar. *Design Recovery* uses a set of fact annotation overrides similar to the technique introduced in Section 3.2 to extract a design fact base of the system. *Hot Spots* has a set of grammar overrides to allow markup (Section 4.5) and transform date sensitive code. While this architecture originated in LS/2000, it is a general purpose analysis and transformation architecture that has been used in many post-Y2K tasks and is still reflected in our research program.

6. Related Work

While this paper concentrates on the use of agile parsing in TXL, the idea is by no means limited to TXL. Any flexible parsing technology that allows for grammar modification can in theory be used to exploit the same techniques.

Van den Brand et al. (van den Brand,1998) have been leading proponents of the need for flexible parsing technology in software engineering, and their ASF+SDF meta-environment (van den Brand,2001, van den Brand,2002) provides a very general flexible parsing and source transformation toolkit. Lämmel (Lämmel 2001b) has already exploited this toolkit to provide a grammar adaptation system with strong similarities to our agile parsing. Cox (Cox,2000) notes the limitations of parsing in software engineering tasks and proposes providing a different kind of flexibility using a robust lexically-based lightweight method to simplify analysis by avoiding a full parse. Generalized parsing techniques are also exploited in DMS (Baxter 1997).

The advantages of programs as structured documents and the importance of XML markup in program comprehension tasks has been well documented by Badros (Badros 2000), Cox (Cox 2000), Maletic (Maletic 2002) and many others. Badros (Badros 2000) discusses the limitations of AST markup produced by traditional parsers, and proposes a powerful special-purpose XML markup for Java. Maletic (Maletic 2002) discusses the advantages of XML source markup in viewing and understanding programs. More details about the use of agile technique for markup are discussed in detail by Cordy (Cordy 2003).

The need to reflect results of program comprehension into source is well discussed in our own papers on HSML (Cordy 2001) and LS/2000 (Dean 2001). Island parsing techniques in source code analysis were first described by van Deursen and Kuipers (van Deursen, 1999) and further discussed by Moonen (Moonen 2001, Moonen 2002). The idea of structuring grammars to make rulesets easier to author and understand has been discussed by Sellink and Verhoef (Sellink 1998, Sellink 1999).

Our paper extends this body of work by showing how grammars can be effectively extended on an *ad hoc* basis to assist in each individual analysis and transformation task. Our experience in industrial application of TXL has shown that it is just as effective to tune the grammar for performance and simplicity as it is to tune the code.

7. Conclusions

The flexible grammar definition capabilities and efficient parsers of modern generalized parsing techniques make it easy for programmers to customize a grammar to the problem. This leads to a new paradigm for programming software comprehension systems - *agile parsing*. Using agile parsing, grammars are customized to provide a parse of the source which is most convenient or efficient for each particular analysis or transformation task. Often this customization can significantly reduce and simplify the analysis and transformation rules of the task.

The TXL programming language explicitly supports agile parsing using the concept of *grammar overrides*. Grammar overrides allow separate specification of custom modifications to the input language's base grammar suitable for each analysis tool without disturbing or cloning the original grammar. TXL's polymorphic grammar capabilities and ordered ambiguity resolution make it particularly well suited to agile parsing.

As modern generalized rewriting systems such as TXL, DMS and ASF+SDF are used for more and more industrial projects involving large scale legacy software systems, techniques that make the best use of the strengths of these systems becomes more important. By leveraging on the capabilities of modern general parsers to simplify and generalize analysis and transformation rules, agile parsing has an important role to play in these applications.

Bibliography

Badros, G. "JavaML: a Markup Language for Java source code", *Computer Networks* 33,1-6 (June 2000), pp. 159-177.

Bell Canada, *Datrix Abstract Semantic Graph: Reference Manual, version 1.4*, Bell Canada Inc., Montreal Canada, May 01, 2000.

van den Brand, M., Sellink, A., and Verhoef, C. 1998. Current Parsing Techniques in Software Renovation Considered Harmful. Proc. 6th International Workshop on Program Comprehension (IWPC 98). Ischia, Italy, pp. 108–117.

van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, C., and Visser, J., 2001. The ASF+SDF Meta-Environment: a component-based language development environment. Compiler Construction 2001 (CC 2001), Lecture Notes in Computer Science, R. Wilhelm, ed., Vol 1827, Springer Verlag, pp. 365–370.

van den Brand, M., Heering, J., Klint, P., and Olivier, P. 2002. Compiling Rewrite Systems: The ASF+SDF Compiler. ACM Transactions on Programming Languages and Systems. 24(4) (July 2002):334–368.

van Deursen, A., and Kuipers, T. 1999. Building Documentation Generators. Proc. International Conference on Software Maintenance (ICSM 99). Oxford, England, pp. 40–49.

Baxter, I.D., and Pidgeon, C.W. 1997. Software Change Through Design Maintenance. Proc. 1997 International Conference on Software Maintenance. Bari, Italy, pp. 250–259.

Cordy, J.R., Halpern, and C.D., and Promislow, E., 1991. TXL: A Rapid Prototyping System for Programming Language Dialects. Computer Languages, 16(1):97-107.

Cordy, J.R., Carmichael, I.H. and Halliday, R., 2000. The TXL Programming Language - Version 10, Kingston: Queen's University at Kingston and Legasys Corporation.

Cordy, J., Schneider, K., Dean, T., and Malton, A., 2001. HSML: Design Directed Source Code Hot Spots. Proc. 9th International Workshop on Program Comprehension(IWPC 01), Toronto, Canada, pp. 145–154.

Cordy, J., Dean, T., Malton, A., and Schneider, K., 2002. Source Transformation in Software Engineering using the TXL Transformation System. Special Issue on Source Code Analysis and Manipulation, Journal of

Information and Software Technology 44(13):827-837.

Cox, A. and Clarke, C., 2000. A Comparative Evaluation of Techniques for Syntactic Level Source Code Analysis. Proc. 7th Asia-Pacific Software Engineering Conference (APSEC'00), Singapore, pp. 282-291.

Dean, T., Cordy, J., Schneider, K., and Malton, A. 2001. Experience Using Design Recovery Techniques to Transform Legacy Systems. Proc. International Conference on Software Maintenance (ICSM 2001), Florence, Italy, pp. 622-631.

Guo, X., Cordy, J., and Dean, T., "Unique Renaming of Java Code Using XML", in preparation.

Johnson, S.C., 1975. Yacc: Yet Another Compiler-Compiler. Computing Science Technical Report No. 32. Bell Laboratories, Murray Hill, N.J.

Lamb, D., and Schneider, K. 1992. Formalization of Information Hiding Design Methods. Proc. IBM Center for Advanced Studies Conference (CASCON '92), Toronto, Canada, pp. 201-214.

Lämmel, R., and Verhoef, C., 2001a. Semi-automatic Grammar Recovery. Software Practice & Experience. 31(15):1395-1438.

Lämmel, R.. 2001b. Grammar Adaptation. Proc. International Symposium on Formal Methods Europe (FME'01), Berlin, pp. 550-570.

Lethbridge,T., Plödereder, E., Tichelaar, S., Riva, C., and Linos, P. 2001. *The Dagstuhl Middle Model (DMM)*, Version 0.003, http://scgwiki.iam.unibe.ch:8080/Exchange/2.

Lesk, M.E., and Schmidt, E. 1975. Lex – A Lexical Analyzer Generator. Computing Science Technical Report No. 39. Bell Laboratories, Murray Hill, N.J.

Malton, A.J., Schneider, K.A., Cordy, J.R., Dean, T.R., Cousineau, D., and Reynolds, J. 2001. Processing Software Source Text in Automated Design Recovery and Transformation. Proc. 9th International Workshop on Program Comprehension (IWPC 2001), Toronto, Canada, pp. 127-134.

Moonen, L. 2001. Generating Robust Parsers using Island Grammars. Proc. 8th Working Conference on Reverse Engineering (WCRE 01). Stuttgart, Germany, pp 13-22.

Moonen, L. 2002. Lightweight Impact using Island Grammars. Proceedings 10th International Workshop on Program Comprehension (IWPC 02), Paris France, pp 343-352.

Müller, H., and Klashinsky, K. 1988. Rigi – A System for Programming-in-the-Large. Proc. 10th International Conference on Software Engineering (ICSE 88), Singapore, pp. 80-86.

Neighbors, J., 1984. The Draco Approach to Constructing Software from Reusable Components. IEEE Transactions on Software Engineering, 10(5):564-574.

Reasoning Systems, 1992. Refine User's Manual, Palo Alto, California.

Sellink, A., and Verhoef, C.. 1999. An Architecture for Automated Software Maintenance. Proc. 7th International Workshop on Program Comprehension (IWPC 99), Pittsburgh, Pennsylvania, pp. 38–48.

Sellink, A., Vand erhoef, C., 1998. Native Patterns. Proc. 5th Working Conference on Reverse Engineering, Honolulu, Hawaii, pp. 89-103.

TXL Project, 2003. FreeTXL 10.2e. http://www.txl.ca/ndownload.html.

Visser, E. 2001. "Stratego: A Language for Program Transformation Based on Rewriting Strategies. System description of Stratego 0.5. Rewriting techniques and Applications (RTA '01), Lecture Notes in Computer Science, A Middeldorp, ed. SpringerVerlag, pp. 357–361.